Chapitre 6

Compléments sur les fonctions - Modules

Nous avons déjà acquis un certain nombre de connaissances sur le langage Python.
La notion de fonction y est fondamentale, comme dans tout langage de programmation procédural.

Dans ce chapitre « théorique », nous allons revenir sur certaines choses qui ont volontairement été laissées de côté dans les chapitres précédents, mais que tout programmeur Python se doit de connaître.

6.1 Fonctions et paramètres

6.1.1 Signature d'une fonction

Nous avons déjà vu la fonction `round` qui permet d'arrondir un nombre. Combien cette fonction prend-elle de paramètres ?

```
>>> round(3.1415, 2) # 2 paramètres ici
3.14
>>> round(3.1415)    # 1 seul paramètre ici
3
```

En consultant le « mode d'emploi » de cette fonction, on peut lire que le second paramètre est optionnel, par défaut égal à 0 :

```
>>> help(round)
Help on built-in function round in module builtins:

round(number, ndigits=None)
    Round a number to a given precision in decimal digits.

    The return value is an integer if ndigits is omitted or None.  Otherwise
    the return value has the same type as the number.  ndigits may be negative.
```

On y découvre également la **signature** de la fonction `round` :

```
round(number, ndigits=None)
```

Il faut savoir déchiffrer une signature. Dans l'exemple précédent, on peut lire :

- Le nom de la fonction : `round`
- Le nombre de paramètres utilisés, ici :
  - Le paramètre `number`, obligatoire : c'est le nombre à arrondir
  - Le paramètre `ndigits`, **facultatif car possédant une valeur par défaut** : c'est le nombre de chiffres après la virgule souhaité. Par défaut, il est égal à `None`. Sans ce paramètre, ou si défini à `None`, la fonction retourne l'entier le plus proche de `number` : "*The return value is an integer if ndigits is omitted or None*"

6.1.2 Paramètres optionnels

Pour définir un paramètre optionnel dans une fonction, il suffit de renseigner sa valeur dans la définition de la fonction :

```
1 def direBonjour(prenom = ""):       # Par défaut, le prénom est la chaîne vide
2   if prenom == "":                  # Si le prénom est vide, on affiche une phrase générique
3     print("Bonjour cher visiteur.")
4   else:                             # Sinon, on dit bonjour au prénom passé en paramètre
5     print("Bonjour", prenom)
```

```
>>> direBonjour()
Bonjour cher visiteur
>>> direBonjour("Jean")
Bonjour Jean
```

Si on ne donne pas de valeur par défaut, le paramètre est alors obligatoire.
Il est tout à fait possible de mélanger paramètres optionnels et obligatoires dans une même fonction :

```
1 def histoire(animal, adjectif = "petit", hobbie = "s'amuser"): # Seul animal est obligatoire
2   print("C'est l'histoire d'un", adjectif, animal, "qui aimait", hobbie)
```

```
>>> histoire("ours")
C'est l'histoire d'un petit ours qui aimait s'amuser
>>> histoire("loup", "grand méchant", "manger")
C'est l'histoire d'un grand méchant loup qui aimait manger
```

▷ ***Peut-on appeler la fonction précédente sans donner l'adjectif, mais en donnant le hobbie ?***

Essayons :

```
>>> histoire("oiseau", "voler")
C'est l'histoire d'un voler oiseau qui aimait s'amuser
```

Sans surprise, Python a remplacé `animal` par *oiseau*, `adjectif` par *voler* et `hobbie` par *s'amuser*, sa valeur par défaut.
Cependant, il est possible en Python de passer les paramètres à la fonction **dans n'importe quel ordre, à condition de les déclarer par leurs noms** :

```
>>> histoire(animal = "oiseau", hobbie = "voler")
C'est l'histoire d'un petit oiseau qui aimait voler
>>> histoire(adjectif = "gros", animal = "poisson")
C'est l'histoire d'un gros poisson qui aimait s'amuser
```

Question 01 Définir une fonction `demander_cafe(type)` où `type` est un paramètre optionnel, fonctionnant sur le principe suivant :

```
1 >>> demander_cafe()
2 Un café s'il vous plaît !
3 >>> demander_cafe("ristretto")
4 Un ristretto s'il vous plaît !
```

Question 02 On considère la fonction `calcul` suivante :

```
1 def calcul(a, b = 1, c = 1):
2   return a + b + c
```

Déterminer le résultat des instructions suivantes :

1. `calcul(1,2,3)`
2. `calcul(4,5)`
3. `calcul(0)`
4. `calcul()`
5. `calcul(a = 2, c = 3)`
6. `calcul(b = 4, c = 1)`

6.2 Portée des variables

En Python, comme dans la plupart des langages, on trouve des règles qui définissent la **portée des variables**. S'interroger sur la portée d'une variable, c'est se poser la question suivante :

Quand et comment une variable est-elle accessible ?

Il est normal que ce genre de questions ne vienne pas directement à l'esprit, tout comme il est normal d'y répondre.

Espace local - Espace global

Considérons la fonction suivante :

```
1 def ma_fonction():
2   a = 2
3   print("Coucou !")
```

Depuis la console, on appele alors la fonction. Puis on demande la valeur de `a` :

```
>>> ma_fonction()
Coucou !
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

▷ ***`a` n'est pas définie ? Pourtant, on a appelé la fonction dans laquelle se trouve l'instruction `a = 2` !***

`a` a bien été définie, mais **dans l'espace local de la fonction**, et non dans l'**espace global** qui constitue notre programme. Elle est inaccessible depuis l'extérieur de la fonction.
Cependant, si on la définit avant la fonction, tout va bien, ou presque :

```
1 a = 4
2
3 def ma_fonction():
4   a = 2
5   print("Coucou !")
```

```
>>> a
4
>>> ma_fonction()
Coucou !
>>> a
4
```

▷ ***Ce n'est pas normal ! Après l'appel de `ma_fonction`, la valeur de `a` aurait dû changer ! Magie noire !***

Bien au contraire ! **Heureusement** que la valeur de `a` n'a pas été modifiée !
Pour bien comprendre, prenons le programme suivant :

```
1 heures = 6
2 minutes = 37
3 secondes = 22
4
5 print(conversionSecondes(heures, minutes, secondes))
```

Le programme affiche :

```
23842
```

La fonction `conversionSecondes` est une fonction récupérée honteusement sur Internet. Nous ne l'avons pas écrite nous même, mais voici son contenu :

```
1 def conversionSecondes(h,m,s): # Conversion en secondes d'une durée h:m:s
2   secondes = h*3600 + m*60 + s
3   return secondes
```

▷ ***Et alors ? Où est le problème ?***

Le problème est le suivant : dans la fonction `conversionSecondes`, il y a une variable `secondes`, comme dans notre programme. De quel droit une fonction écrite par quelqu'un d'autre pourrait modifier la valeur de notre variable `secondes` ?

L'incidence peut paraître minime, mais imaginez que la variable `secondes` rentre en jeu dans le déclenchement d'un réveil ou d'une alarme. L'appel de la fonction `conversionSecondes` modifierait sa valeur en la passant de 22 à ... 23842 !

Depuis un espace local, comme dans une fonction, on ne peut pas modifier la valeur d'une variable numérique définie hors de cet espace local.

Question 03 On considère le programme suivant. Que va afficher la dernière instruction ?

```
1 a = 3
2
3 def affichage():
4   a = 5
5   return a
```

```
>>> fonction()
5
>>> a
...
```

▷ ***OK... Une fonction ne peut pas modifier une variable « extérieure ». Mais peut-elle accéder à sa valeur ?***

Bien sûr ! D'ailleurs, nous l'avons déjà fait :

```
1 from math import pi
2
3 def aireDisque(r): # Retourne l'aire du disque de rayon r
4   return pi*r**2
```

```
>>> aireDisque(3)
28.274333882308138
```

`pi`, qui provient du module `math`, est bien extérieur à la fonction !
On peut changer sa valeur, mais la modification ne se fera que dans l'espace local de la fonction.

```
1 from math import pi
2
3 def aireDisque(r): # Retourne l'aire (fausse) du disque de rayon r
4   pi = 2.56        # On modifie la valeur de pi
5   return pi*r**2
```

```
>>> aireDisque(3)
23.04
>>> pi
3.141592653589793
```

Le résultat de `aireDisque` est alors faux, car `pi` a été modifié dans la fonction, mais pas en dehors.

6.3 Modules

Terminons ce chapitre par quelques compléments sur les modules.

6.3.1 Espaces de noms

Nous avons vu que pour importer des constantes et des fonctions depuis un module, on utilise la syntaxe :

```
from ... import ...
```

Pour utiliser le nombre mathématique π, on l'importe depuis le module `math` de la manière suivante :

```
from math import pi
```

On peut également tout importer depuis le module `math` en faisant :

```
from math import *
```

Mais imaginons qu'après l'importation du module, on modifie - consciemment ou non - la valeur de la variable `pi`. Tous nos calculs seraient faussés !

▷ ***En même temps, quelle idée...***

En effet... Autre exemple :

```
1 from supermodule import *
2
3 def calculer(a,b):
4   return a + b - a*b
5
6 resultat = calculer(3,4)
7 resultatFinal = superFonction(resultat)
8
9 print(resultatFinal)
```

`supermodule` est un module imaginaire contenant notamment la fonction `superFonction`. Nous ne l'avions pas remarqué, mais dans `supermodule`, il existe également une fonction `calculer`, utilisée notamment dans la fonction `superFonction`. Et nous avons re-définit cette fonction...

Pour éviter ce genre de désagréments, il est possible d'importer des constantes et des fonctions depuis un module en y ajoutant un **espace de noms**. On utilise la syntaxe `import nomDuModule`, de la manière suivante :

```
import math
```

En important le module `math` de cette manière, il est alors nécessaire de précéder toute constante ou fonction du module par `math.` :

```
>>> math.pi
3.141592653589793
>>> math.sqrt(2)
1.4142135623730951
```

Même en définissant une variable `pi`, il n'y aura plus de conflit :

```
>>> pi = 3.14
>>> pi
3.14
>>> math.pi
3.141592653589793
```

6.3.2 Création de module

Le plus compliqué pour créer un module, c'est de savoir ce qu'on va écrire à l'intérieur. Le reste est très facile.

Question 04 Dans le dossier de votre choix, créez deux fichiers Python depuis votre éditeur Python favori :

- un fichier *main.py*, qui constituera notre programme
- un fichier *fonctions.py*, qui constituera notre module

Question 05 Dans *fonctions.py*, écrire les fonctions suivantes :

1. la fonction `maxi(a,b)` qui retourne le maximum des deux nombres `a` et `b`
2. la fonction `mini(a,b)` qui retourne le minimum des deux nombres `a` et `b`
3. la fonction `moyenne(a,b)` qui retourne la moyenne des deux nombres `a` et `b`

Question 06 Depuis *main.py*, importer le module avec la commande `import fonctions` (ne pas écrire *.py*).
Définir ensuite, à l'aide des fonctions du module `fonctions`, la fonction `moyenne3(a,b,c)` qui retourne la moyenne des deux plus grandes valeurs parmi les nombres `a`, `b` et `c`.